Slicing of Object-Oriented and Aspect-Oriented Programs

Santosh Kumar Behera

Roll. 212cs3369

under the guidance of

Prof. Durga Prasad Mohapatra

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela – 769 008, India

Slicing of Object-Oriented and Aspect-Oriented Programs

Dissertation submitted in

May 2014

to the department of

Computer Science and Engineering

of

National Institute of Technology Rourkela

in partial fulfillment of the requirements

for the degree of

Master of Technology

by

Santosh Kumar Behera

(Roll. 212cs3369)

under the supervision of

Prof. Durga Prasad Mohapatra

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela – 769 008, India

Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, India. www.nitrkl.ac.in

Dr. Durga Prasad Mohapatra
Professor

May 23, 2014

Certificate

This is to certify that the work in the thesis entitled *Slicing of Object-Oriented and Aspect-Oriented Programs* by *Santosh Kumar Behera*, bearing roll number 212CS3369, is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of *Master of Technology* in *Computer Science and Engineering Department*. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Durga Prasad Mohapatra prof. , CSE Dept. NIT Rourkela

Acknowledgment

First of all, I would like to express my deep sense of respect and gratitude towards my supervisor Prof. Durga Prasad Mohapatra, who has been the guiding force behind this work. I want to thank him for introducing me to the field of Program Slicing and giving me the opportunity to work under him. His undivided faith in this topic and ability to bring out the best of analytical and practical skills in people has been invaluable in tough periods. Without his invaluable advice and assistance it would not have been possible for me to complete this thesis. I am greatly indebted to him for his constant encouragement and invaluable advice in every aspect of my academic life. I consider it my good fortune to have got an opportunity to work with such a wonderful person.

I thank our H.O.D. Prof. Santanu Kumar Rath and Prof. Durga Prasad Mohapatra for their constant support in my thesis work. They have been great sources of inspiration to me and I thank them from the bottom of my heart.

I would also like to thank all faculty members, PhD scholars, my seniors and juniors and all colleagues to provide me their regular suggestions and encouragements during the whole work.

At last but not the least I am in debt to my family to support me regularly during my hard times.

I wish to thank all faculty members and secretarial staff of the CSE Department for their sympathetic cooperation.

Santosh Kumar Behera

Abstract

Program slicing[1] has many applications in a software development environment such as debugging, testing, anomaly detection, program understanding and many more. The concept being introduced by Weiser and it was started with static slicing calculation[1]. Talking about static slicing, it is a subset of statements of a program which directly or indirectly affect the values of the variables computed providing a slicing criterion. Dynamic slicing[3] is the counterpart of the static slicing i.e finding the statements which are really affected by giving the particular input value of the variable. Object-Oriented Program(OOP) has been the most widely used software development technique. OOP is still popular among many companies for their product development.There are some drawbacks of the OOP implementation. One of them is cross-cutting concerns. Aspect-Oriented Program[23] provides separation of cross-cutting concerns from the core modules by introducing a new unit of modularization, called Aspect. In this project, we have developed an Approach which creates System dependence Graph(SDG)[2] which is the intermediate representation of an OOP and AOP, then takes that SDG as input to compute the slice of that program with respect to slicing criterion.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Program slicing[1] is the method of computing subset of statements of a program that may affect the values at some point of interest, can be referred to as a slicing criterion. Program slicing is an important technique that has been implemented in the field of software development with applications such as debugging, testing, understanding of complicated codes, anomaly detection and many others. Program slices can be of many types such as static slicing, dynamic slicing, forward slicing, backward slicing[9] etc. Talking about static and dynamic slices, static slicing computes slices without considering the program input whereas dynamic slicing consider program input values . It contains only those statements that actually affect the value of a variable. Now to represent the input program For which we have to calculate slices with respect to given slicing criterion, we can use various kinds of graphs such as control flow graph (CFG), program dependence graph (PDG), system dependence graph (SDG)[2] and many more depending upon the type of input program. For example, if we take a program that consist of only a single function with no other function call, then a PDG can be used as an intermediate representation. Similarly for an input program having function calls along with a main function or to say an Inter-procedural program, we can use SDG. This paper consider SDG approach for representing an input program and applying different algorithm to compute slice of object-oriented program as well as Aspect-oriented programs. In this paper we

develop an Approach which generates SDG of Object-Oriented program(OOP) as well as Aspect-Oriented program(AOP) and then compute slice according to the slicing criterion.

1.1 Motivation

Now a days the kind of software we are used in this computer world are very large in size as well as complex in nature so that it is very difficult for understanding, maintaining, testing and debugging the code. Mainly In order to find the bugs in the program we are searching whole program line by line which is very much difficult and also time taking task. To resolve these issues Mark wiser introduced an approach i.e. program slicing which finds the interdependence statements from the program.

In all the slicing algorithms so far proposed by different researchers are about OOP and very few are about of AOP. They have taken SDG as intermediate graph representation for slicing but it is not stated clearly about how SDG created i.e. not automated which takes more time in the process of compute slicing.

1.2 objective

Our main Objective of research work is to develop efficient slicing Algorithms. To address this broad objective, we identify the following goals:
We wish to compute slices of Object-Oriented and Aspect-Oriented programs as quick as possible. To fulfillment of this we plan to

- Construct a System Dependence Graph, which is the intermediate representation of Object-Oriented and Aspect-Oriented Programs
- Slice the Object-Oriented and Aspect-Oriented Programs by using the SDG.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows:

1. Chapter 1: In this chapter we have discussed about the introduction to Program Slicing, motivation and objective of our research.

2. Chapter 2: In this chapter we have discussed the basic concepts which are useful in our research.

3. Chapter 3: In this chapter we present the literature review where we have described some existing works on Program Slicing.

4. Chapter 4: In this chapter we present our proposed Approach for Slicing of Object-Oriented Program.

5. Chapter 5: In this chapter we present our proposed Approach for Slicing of Aspect-Oriented Program.

6. Chapter 6:At last we concluded in this chapter..

Chapter 2

Basic Concepts

In the field of program slicing[1] lots of basic concepts have been used by several researchers. In our innovative world many new concepts have been introduced by different researchers. For extending program slicing, many algorithms have been proposed to enhance the efficiency of program slicing. This chapter provides definition and detailed explanation with examples of several basic concepts. This chapter also mentions many more applications of program slicing.

2.1 Types of Dependencies

There are different types of dependencies present in any program, following are the detail explanation about that dependencies

- **Data dependence[1]**: In this dependence the program statement depends on the data of the preceding statements of the program.
- **Control dependence[1]**: A statement is control dependence on the preceding statement if the out come of that preceding statement determines whether the former statement should be executed or not.

The following example shows the data dependence and control dependence:

```
s1.   main ( )
      {
s2.         int  x, y,z;
s3.         x=10;
s4.         y=20;
s5.         If(x>y)
s6.               cout<<" the largest value is"<< x;
          else
s7.               cout<<" the largest value is"<< y;
      }
```

Figure 2.1: *Example of Data dependence and Control dependence*

From the above program statement s6 is data dependant on the statement s3 and the statement s7 is control dependant on the statement s5.

2.2 Program Representation[2]

Normally if we slice the program directly from the code, is not that much of easy, so make easy to understand and to slice many researchers have been proposed different intermediate representation of the program. Then applying different slicing algorithm on this intermediate representation to compute the slice according to the given slicing criterion. There are many different types of intermediate Graph representation but here in this section explain the detail about Program dependency Graph (PDG) and System dependence Graph (SDG).

2.2.1 Program dependency Graph (PDG)

A PDG[2] explicitly represents both control and data dependencies in a single program representation. A PDG representation of a program is a graph in which the nodes represent the statements, and the edges represent inter- statement data or control dependencies. The program dependence graph G of a program P is defined as the graph G= (N, E), where each n ∈ N represents set of statement of the program P and E represents set of edges that shows the dependence among the statements.

PDG contains two kind of directed edges: **Control dependence edge** and **Data dependence edges**.

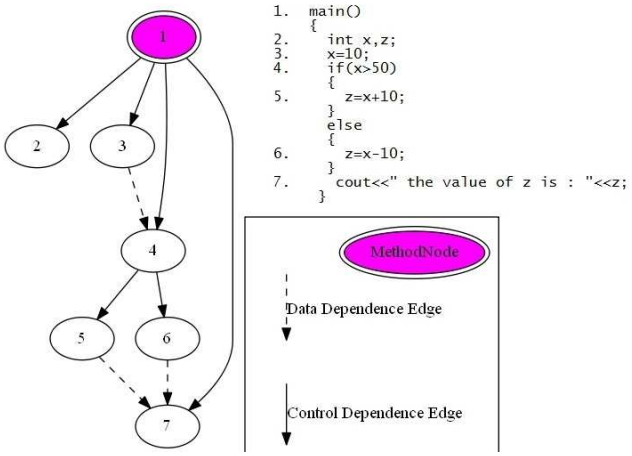

Figure 2.2: *Program Dependence Graph*

The above shows the PDG of the program mentioned in bellow and also shows the control dependencies and data dependencies within this PDG.

2.2.2 System Dependence Graph(SDG)

In PDG there is no representation of procedure call, i.e. the PDG cannot handle the procedure calls. Hence the PDG helps for those program having only one procedure only. So the System Dependences Graph (SDG)[2] is the another representation of the program in which we can Represent all the feature of PDG as well as can handle procedure call. i.e. we can say SDG is the collection of number of PDGs. The technique for construction of an SDG is that of first construction a PDG for every method, including the main method and then adding extra dependence edges which connects the various sub-graphs together.

Mainly PDGs are added with the following kinds of the new edges to construct SDG

- **call edge**[6,7]: Edge is added from call site vertex to corresponding procedure entry vertex.

- **parameter-in edge**[6,7] : Edge is added from each actual in vertex to corresponding formal-in vertex.
- **parameter-out edge**[6,7] : Edge is added from each formal-out vertex to corresponding actual-out vertex.

2.3 Program Slicing

For testing and maintaining the software, we need to reduce the complexity of the program. For that purpose the whole program need to be broken into sub-parts. Program slicing helps in this regards. Program slicing was first introduced by Weiser[1] in 1979. It is an analysis and transformation technique which uses the dependency relationship among the statements for identifying the part of a program that affects or gets affected by appoint of interest. This point of interest is known as the slicing criteria. Subset of the program statements those influence or get influenced by the variable which is given in the slicing criteria are added to the slice. Generally, for the construction of program slice we need to define a slicing criterion. A slicing contains a set $< s, v >$, where s denotes the statement number and v denotes the variable of program[1].

There are many types of program slicing and bellow we have discussed some of them

2.3.1 Forward Slicing[9]

It computes all those parts that might be affected by the slicing criterion, using their dependence on the slicing criterion.

2.3.2 Backward Slicing[9]

It contains all the statements of the program those might have affected the variable at the statement given in the slicing criterion. Following examples explain the forward

and backward slicing.

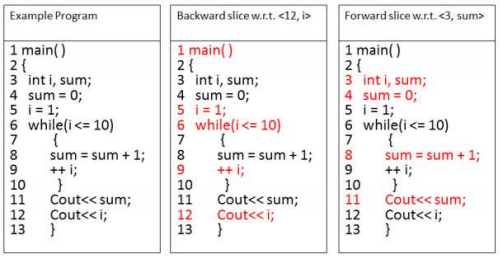

Figure 2.3: *forward and backward slicing*

2.3.3 Static slicing

Static slicing[9] is used to identify those statements of the program that potentially contribute to the computation of the selected statements for all possible programs inputs. Static slicing helps to understand of these parts of the program that contribute to the computation to the selected function. As we know the static slicing has many advantages in the process of program understanding, still it has large subprograms because of the imprecise computation of these slices. In addition, static slices cannot be very useful in the process of understanding of program execution

2.3.4 Dynamic slicing

Dynamic slicing[9] is used to identify those statements of the program that contribute to the computation of the selected statements for a given program execution (program input). Dynamic slicing may help to reduce the size of imprecise computation of the static slice i.e. the part of the program that contributes to the computation of the function of interest for particular program input. Mainly size of Dynamic slices are much smaller than the static slices, thus it is be used to understand program execution. Programmers may still have difficulties to

understand the program and its behavior. Slicing Tool is important to divides methods that will support the process of understanding of large software. To understanding of large software systems, it uses an intermediate representation of a program and then compute a slice from the graph. Following examples explain the static and dynamic slicing.

```
1.   main()
      {
2.       int x,z;
3.       cin>>x;
4.       if(x>10)
          {
5.           z=x-10;
          }
        else
            {
6.           z=x+10;
            }
7.        cout<<" the value of z is :"<<z;
       }
```

Figure 2.4: *Static and dynamic*

In the Static slice with criterion $< 7, x >$ are contains the line no. as 1,2,3,4,5,6,7. But In the dynamic slice w.r.t criterion $< 7, x >$ where input value of x is 5 are contains the lines no. as 1,2,3,4,6,7.

2.4 Application of Program Slicing

This section provides various applications by using program slicing. Initially, the program slicing was developed to create automated code decomposition tools. The main objective of these tools was program debugging. Now-a-days the program slicing techniques has been used in different software development processes.

2.4.1 Differencing the programs

Normally, programmers find difficulties to differentiate two programs. So program slicing technology can be used effectively for differentiating two programs. It helps to find all the components of different programs having different behavior and to

produce a program that captures the semantic differences between two programs by comparing the backward slices of the vertices in two dependence graphs. Here, the backward slice is computed by giving a slice criterion.

2.4.2 Debugging

Debugging[24] is a process of finding and reducing the number of bugs, or defects in the program. The problem of finding bugs in a program is always a difficult task. The process to find a bug involves in running the program several times, searching for each line is more time taking. In distributed systems, the problem is more difficult because of different dependencies i.e. control dependencies, data dependencies and also communication dependencies that might lead to additional bugs. Program slicing was originally proposed for observing the process of debugging carried out by programmers. Programmers virtually compute slice while debugging codes which was difficult and time taking also. so program slicing techniques helps to find the subset of statements according to their dependencies from which it is easy to find bugs in an effective way.

2.4.3 Software Maintenance

Software maintenance[24] is a costly process because each changes to a program must consider into many complex dependence relationships in the existing software. The most challenging part in the software maintenance, are to understand various dependencies in an existing software and to make changes to the existing software without introducing new problems, i.e. whether a code change in a program will make any affect to the behavior of other codes of the program. To overcome this problem, it is important to know which variables will be depended on which statements. This problem can be reduced by slicing technique to the software.

2.4.4 Testing

Software maintainers are often used regression testing[24]. Regression testing deals with re-testing of software after changes or modification. Even after the smallest change to a part of code, extensive tests is required because it might involve running a large number of test cases to ensure that no unwanted behavior arises due to the change. This requires to take new test cases along with the existing test cases. Slicing helps to reduce the number of these test cases. While decomposition slicing reduces the need for regression testing on the complement. there may still be a substantial number of tests to be needed on the dependent, independent and changed parts. Slicing can be used to minimize the number of these tests.

2.4.5 Refactoring

Informally Refactoring[24] is defined as the process of improving the design of existing software system. In this case the source code transformation takes place. While changing each transformation is expected to preserve the behavior of system. There is simple example of refactoring is extracting a class or a method from one class to another. Hence for the case of refactoring program slicing makes an important role as it is the finding the subset of statements of program those are related to each other.

2.4.6 Functional Cohesion

Cohesion measures[24] the relatedness of the code within the component. A highly cohesive software component means in which one function cannot be further divided into sub-module. In a good quality of software always maintains high cohesive. Program slicing helps to find statements those are inter-dependencies within a components.

Chapter 3

Literature Survey

In this section, we explain the survey of basic existing papers those are closely related to our work.

Weiser[1] proposed 1st program slicing approach for the Procedural oriented Program. According to Weiser Program Slicing is a method of decomposition that extracts the statements from programs, those are relevant for a particular computation. Program slicing proposed by Wiser is a kind of backward static slicing. In this wise Approach, it is difficult to represent Pointers and Arrays.

Agarwal and Horgan[3] introduced the approach of dynamic slicing. Dynamic Slice is similar to static one but it is constructed with respect to only one execution of the program. It does not include the statements with respect to the slicing criteria for the particular input.

Horwitz et al.[2] developed a system dependence graph (SDG) as an intermediate representation for procedural programs which contain multiple procedures. They proposed a two phase algorithm by using SDG to compute inter procedural static slice. Larsen and Horrold extended the SDG of Horwitz [25]et al. to represent object oriented programs. They include many object oriented features such as class, objects, inheritance, polymorphism etc on SDG.

Mohapatra et al.[6] have developed edge marking and node marking approach for dynamic slicing of OOP. They have used SDG of OOP for slicing. Their approaches

are based on marking and un-marking of the edges on nodes of the graph accordingly when dependency arises.

Zaho[13] proposed an Aspect Oriented System dependence Graph (AOSDG) which is an extension of object oriented SDG. The AOSDG consists of two parts i.e. one Aspect part and another non-Aspect part. They have used a special edge to connect these two parts representing AOSDG.

Braak et al.[26] extended the AOSDG proposed by Zaho. They used two phase algorithm to find static slice of AO program. They have not addressed the dynamic slicing of AOP.

Sahu et al.[19] proposed the extended-ASDG as the intermediate representation of AO program. Their EASDG is different from ASDG of Zaho in two ways. i.e. each point cuts are explicitly represented and weaving process is represented in EASDG. They have used node marking algorithm on the EASDG to compute dynamic slice of AOP.

Chapter 4

Slicing of Object-Oriented Program

4.1 Block Diagram of our Approach

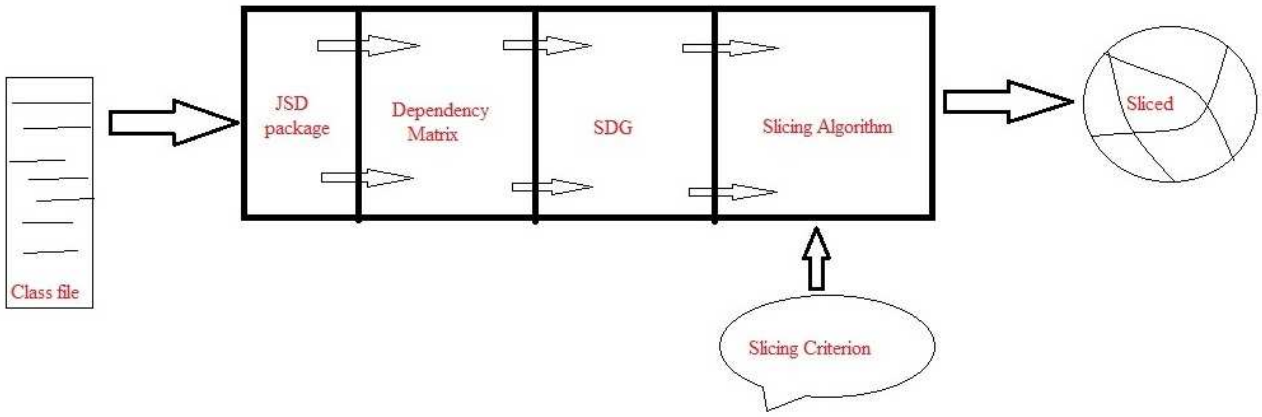

Figure 4.1: *Block Diagram*

The above figure shows actually how our tool works. First we create the class file of the java program to be sliced. Then give that class file to our tool, Then it finds all the dependence matrix using java system dependence package (JSD package)[27]. From that matrix it creates the SDG of that java program. After creating SDG again

we give one slicing criterion, then it uses one slicing algorithm to compute slice of the specified java program.

4.2 Creation of SDG of OOP(Java)

A Java System Dependence Graph[5] is a multi-graph which contains control and data dependencies between the statements of a Java program. it contains classes, methods, statements, interfaces to represent SDG of java program. Each of these represent graph separately and combine with hierarchical manner to make complete SDG of java program. Here the statements are lower level, then method level, like this all are connected in a hierarchical structure within the SDG. Now we discuss the different steps to create SDG.

4.2.1 Statement dependency Graph

Statements are the lowest level in SDG of java program. It is an atomic construct representing a single expression in the source code of the program. A statement representing a call to another method (a callsite) requires a special representation.

4.2.2 Method dependency Graph

It represents a single method or procedure in a program. It is just the next layer from the statement layer. The method entry vertex connects to other members of methods using control dependence edges. Parameter passing is obtained by using actual and formal vertices. The called procedure has formal-in and formal-out vertices, which use parameter variables accordingly. There is a call dependence edge which connects between the call site and the procedure being called.

4.2.3 Class dependency Graph

It represents the classes of the program. It is the next layer to Method Dependency Graph. It contains class entry vertex to connect the method entry vertices by using class member edges. Here, dependent classes are connected by using class dependence edges.

4.2.4 Construct the JSDG

Here we have taken one class named as JavaSytemDependenceGraph to find all the information regarding different dependence as discussed previously. This class contains different linked list for storing different nodes and the dependencies between them. There is a class named as ConvertJsdgToGv which converts Graph using all the information from stored matrix. Finaly we give a specific path to store the SDG of input program

4.2.5 Example

Here we have taken the following example of java program and it creates SDG which looks like the following.

```
1.    public class calc {
2.        int sum(int a , int b)
        {
3.        int c;
4.        c=a+b;
5.        return c;
         }
6.     void max(int x , int y)
        {
7.          int z;
8.          if(x>y)
             {
9.           z=x;
             }
            else
             {
10.          z=y;
             }
11.       System.out.println("Largest between two value is :"+z);
        }
12.       public static void main(String[] args)
        {
13.       int s,t,rt;
14.       calc p= new calc();
15.       System.out.println("enter two value");
16.       Scanner r= new Scanner(System.in);
17.       s=r.nextInt();
18.       Scanner r1= new Scanner(System.in);
19.       t=r.nextInt();
20.       p.max(s,t);
21        rt=p.sum(s,t);
22.       System.out.println("sum of these two value is :"+rt);
        }
    }
```

Figure 4.2: *Java Program*

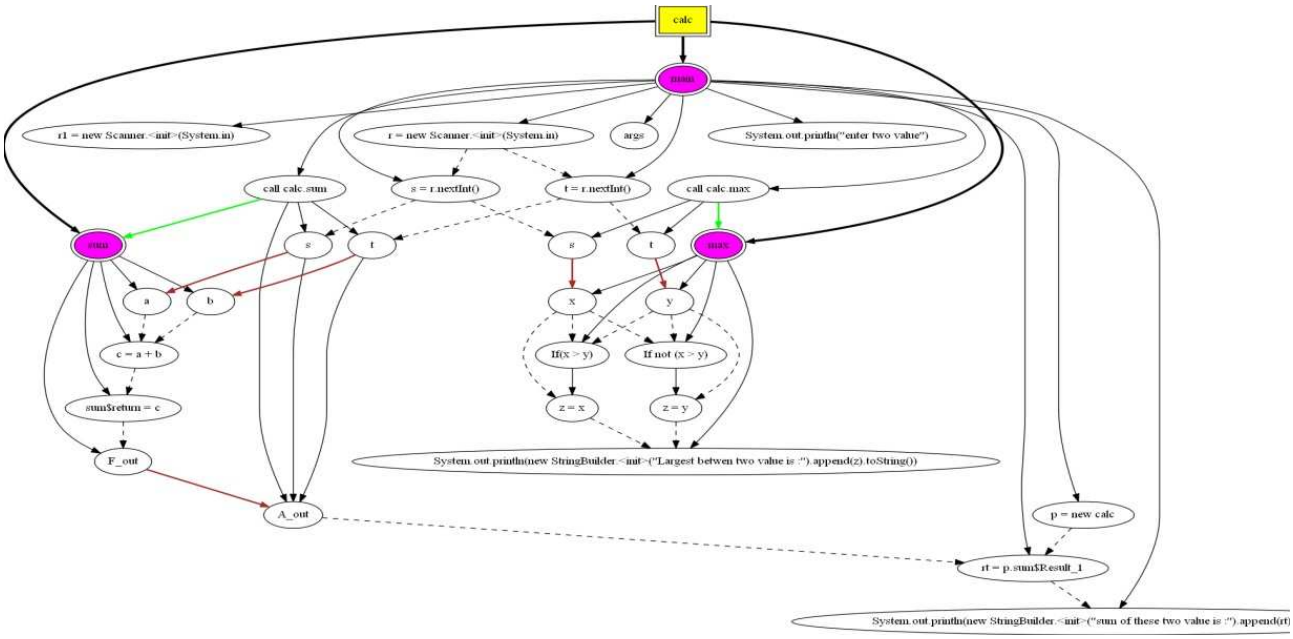

Figure 4.3: *SDG of Java Program*

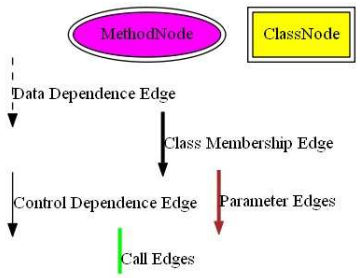

Figure 4.4: *Notation*

4.3 Static Slicing

This section discuss the slicing of the previously generated JSDG. Here we have taken horwitz's[2] two pass algorithm to compute static slice. Our tool takes one input slicing criterion i.e. statement no. show that, it slices using the JSDG by considering that slicing criterion. Now we discuss bellow the Algorithm with taking previous example.

4.3.1 Algorithm

- The traversal in **pass one** start from desired vertex and goes backwards along all edges except parameter-out edges.
- The traversal in **pass two** starts from all the vertices reached in pass one and goes backwards along all edges except call and parameter-in edges.
- The Slice is the union of the two sets of vertices.

Figure 4.5: *Static Slicing Approach*

4.3.2 Case study for the Static slicing

we have taken one Java program shown in figure 4.2 and creates it's SDG as previously discussed which is shown in figure 4.3. now we use static slicing approach i.e. two pass algorithm on this and using $< 22, rt >$ as slicing criterion. then it slices the graph from that point towards up, those lines affects that statement. Figure 4.5 shows the sliced one.

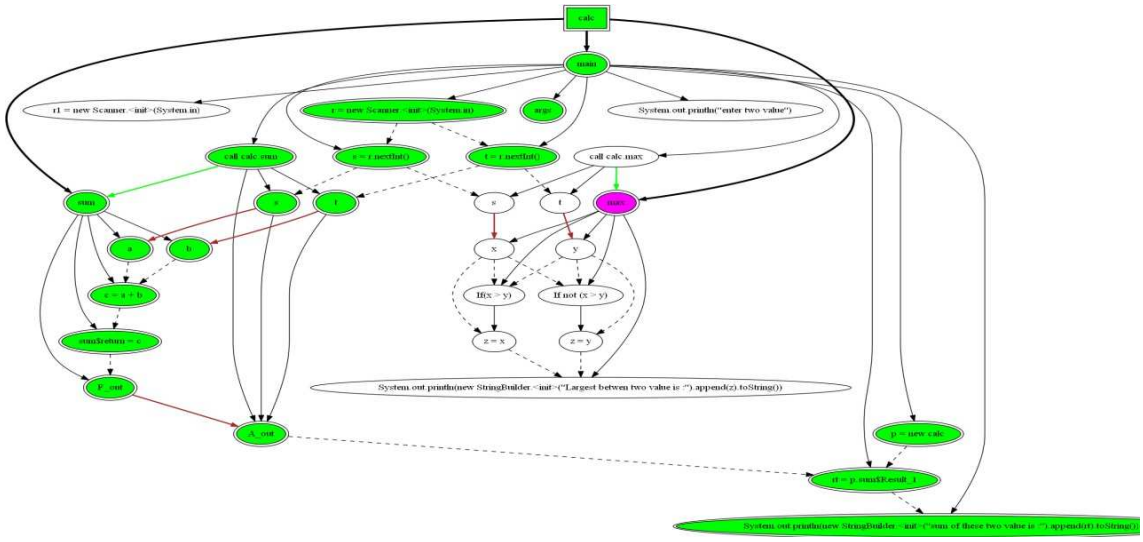

Figure 4.6: *Static Slice with* $< 22, rt >$

4.4 Dynamic Slicing

This section discuss the slicing of the previously generated JSDG. Here we have taken Mohapatra's NodeMarking[6] algorithm to compute dynamic slice. Our tool takes input slicing criterion i.e. statement no. as well as the value of the variable show that, it slices using the JSDG by considering that slicing criterion for that particular input value of the variable. Now we discuss the Algorithm with taking previous example.

4.4.1 Algorithm

1. Construct the SDG of the OOP statically.

2. Do the following before execution of the program P
   - Unmark all the nodes
   - Set *dslice*(u)=Φ for every node u of SDG
   - Set *RecentDef*(var)=Null of each variable

3. Run the program and carry out the following :
   - For each variable var used at u do the following

   Update dslice(u) by using

   dslice(u)={x_1 ,….,x_k} ∪dslice(x_1) …… ∪dslice(x_k)
   - If u is a def(var) node (definition node of var), then
     - Unmark the node RecentDef(var).
     - Update the *RecentDef*(var)=u
   - Mark the node u.
   - If u is a call vertex, then do the following :
     - Mark the vertex u.
     - Mark the corresponding actual_in /out vertices.
     - Mark the method entry vertex of the corresponding called method
     - Mark the corresponding formal_in /out vertices.

4. Exit when execution of program is P is terminated

Figure 4.7: *Node Marking Algorithm*

4.4.2 Case study for the Dynamic Slicing

Now we have taken the same previous example shown in the fig.4.2 and also it creates SDG shown in figure 4.3, then we use the node marking algorithm to compute Dynamic slice. here the slicing criterion is ¡11,z¿ and the input values of x and y are 20,10 respectively.

first the SDG created Statically once, then updated accordingly with the execution trace path, here the execution trace paths are 12,13,14,15,16,17,18,19,20,6,7,8,9,11,21,2,3,4,5,22. then finally it slices with the criterion $< 11, z >$ are 1,6,7,8,9,11,12,13,14,15,16,17,18,19,21 which are shown by shaded in the figure 4.4

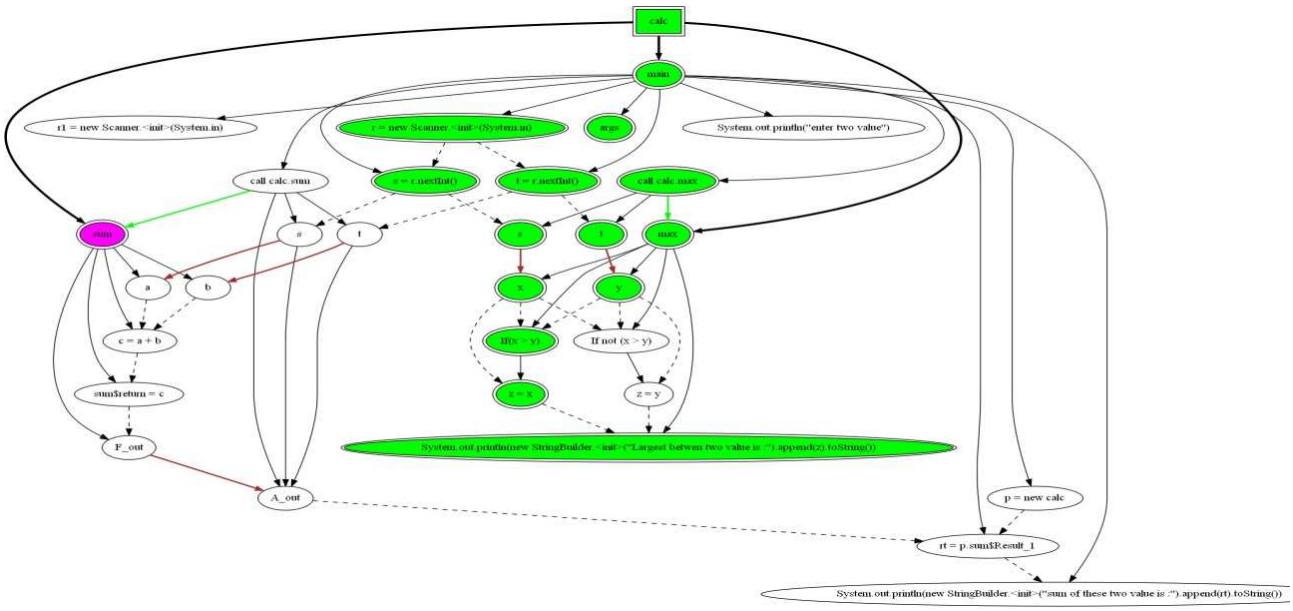

Figure 4.8: *Dynamic Slice with* $< 11, z >$ *x=20,y=10*

4.5 Result

we have taken some of open source example of java program and then compute slicing. The Table 4.1 shown the time taken for generating SDG as well as compute

Slicing.

Table 4.1: Performance

Programs	LOC	Time to generate SDG(ms)	Time to generate Static Slice(ms)	Time to generate Dynamic Slice(ms)
Binary Search Tree	153	110	08	05
Doubly Linked List	133	91	05	03
Elevator	85	50	02	01

Chapter 5

Slicing of Aspect-Oriented Program

5.1 Basic concept of AOP

- There are some draw back of OOP implementation. One of them is Cross-cutting conserns.
- **Cross-Cutting concern** : Cross-Cutting[23] concern is the part of the program Which is scattered across multiple modules of the program.
- Logging is one of the common example of Cross-Cutting concern.
  - Let's say there is an online service provider that provide the following services to its client: pay-per-view TV, magazines and music on demand. The traditional OOP implementation are bellow.

5.1.1 Aspect-Oriented Programming

- The concept of AOP was developed at Xerox PARC, by Gregor Kiczales et al.[23] in the year 1996.

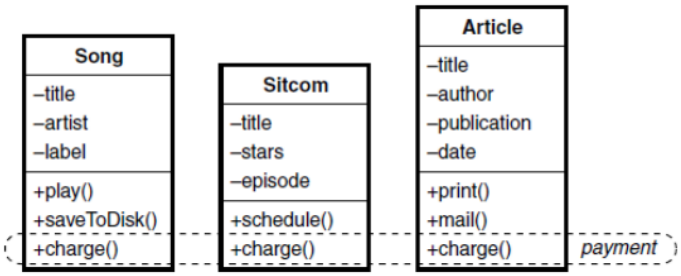

Figure 5.1: *Cross-Cutting*

- AOP provides separation of Cross-Cutting concerns from the core modules by introducing new unit of modularization called Aspect.

5.1.2 Feature of AOP

- **Aspect**: Aspects[23] are like classes in OOP, that contain functionalities.
- **Joinpoints**:Aspects cross-cut[23] object at only well-defined points, such as at object construction,method call or member variable access points. Such well-defined points are known as join points.
- **Pointcut**:The specification for naming a join point is called a Pointcut[23]. Pointcut is the collection of join points.
- **Advice**: Once the join points are spotted in a program, the intended behavioral objective is defined. This behavior is called Advice.

Fig. 5.2, 5.3, 5.4, 5.5, 5.6 are shown all the features of AOP.

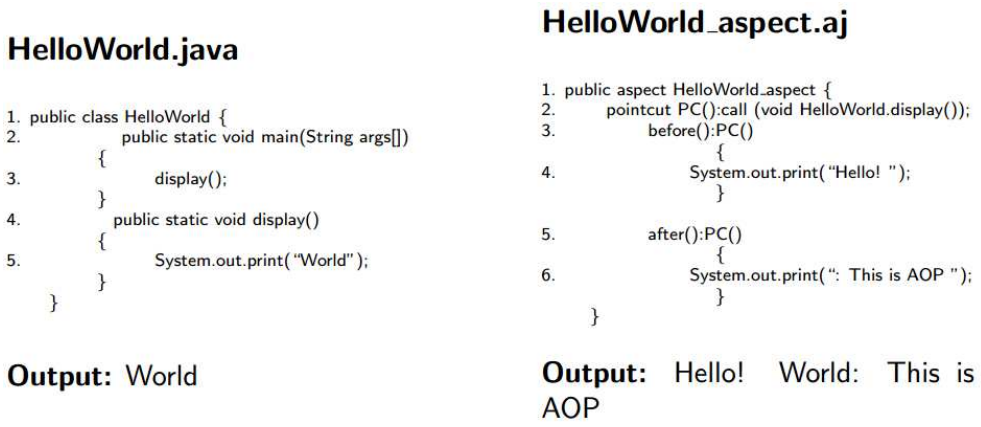

Figure 5.2: *Example of AOP*

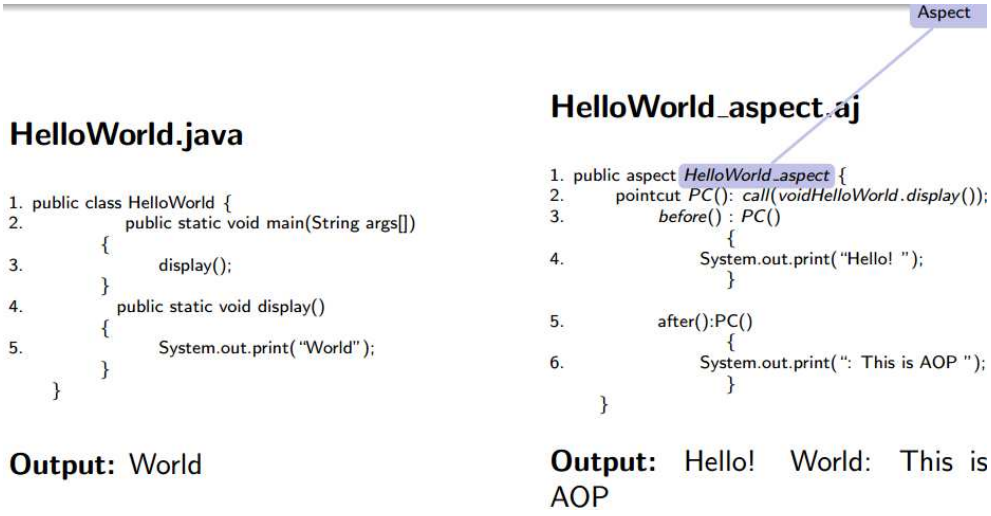

Figure 5.3: *Example of Aspect*

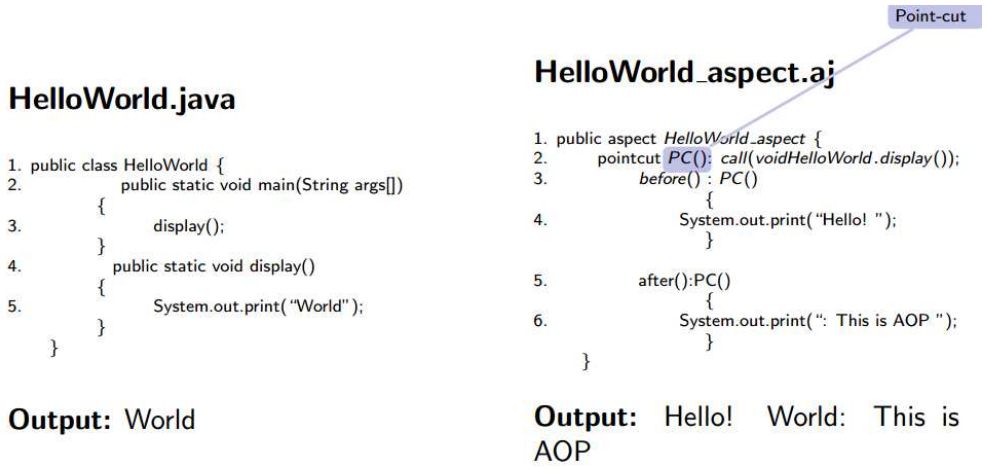

Figure 5.4: *Example of Point-Cut*

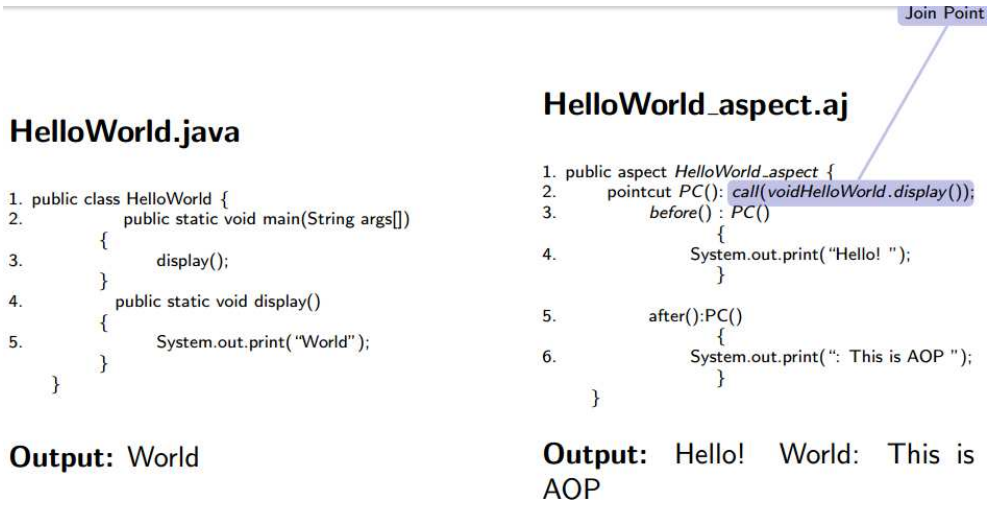

Figure 5.5: *Example of Join Point*

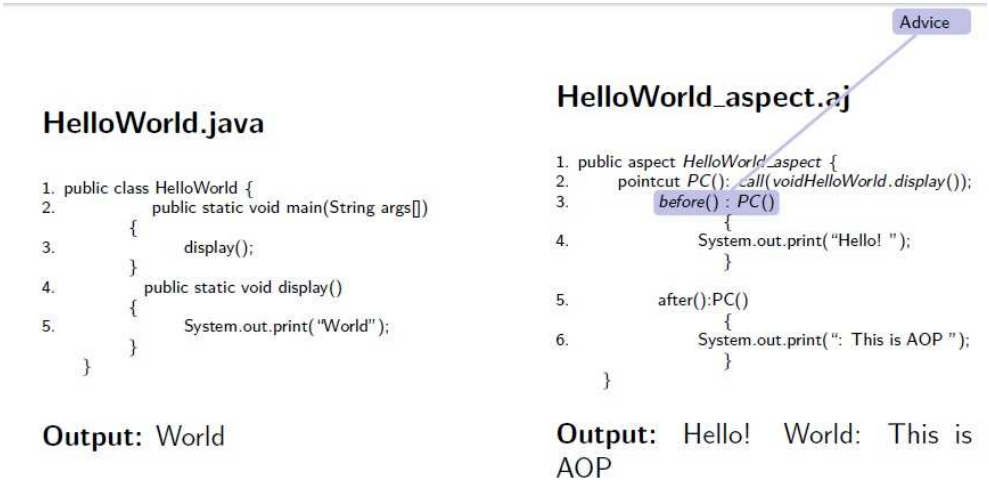

Figure 5.6: *Example of Advice*

5.2 Block Diagram of Tool

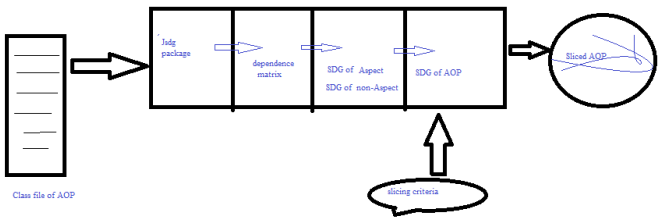

Figure 5.7: *Block Diagram of Tool*

The figure shows actually how our Approach works. First we create the class file of the Java program to be sliced. Then give that class file to our tool, Then it finds all the dependence matrix using java system dependence package[5] (JSD package). From that matrix it creates the SDG of that java program. After creating SDG again we give one slicing criterion, then it uses one slicing algorithm to compute slice of the specified java program.

5.3 Creation of SDG of AOP

- AOP has two part one is the Aspect part and other is the non-Aspect part[23].
- First we generate the SDG of two part separately.
- Create one pointcut table which stores the name of the pointcuts and target methods.
- Using this pointcut table we search the target method for non-Aspect part and then connect to the pointcut node in the Aspect part by an weaving Edge.
- Finaly we got the SDG of input AOP.

5.3.1 Example

Here we have taken the following example of java program and it creates SDG of that AOP which looks like the following.

Account.java

```
a1.   public class Account {
a2.     String name;
a3.     String accno;
a4.     int balance;
a5.     public void deposit(int a)
          {
a6.             balance=balance+a;
          }
a7.     public void withdraw(int b)
          {
a8.             if(balance-b >100)
a9.               balance=balance-b;
                else
a10.              System.out.println("minimum amount reached");
          }
a11     public static void main(String arg[])
            {
a12             Account A= new Account();
a13             A.deposit(2000);
a14             A.withdraw(500);
            }
        }
```

Figure 5.8: *Non-Aspect part of AOP*

log_aspect.aj

```
x1    public aspect log_aspect {
x2    pointcut depositlog(int amnt):call(void Account.deposit(int))&&args(amnt);
x3    before( int amnt): depositlog(amnt)
      {
x4        try{
x5              PrintWriter writer = new PrintWriter ("D:\\SDG\\log.txt");
x6               writer.println("deposit amount="+amnt);
x7               writer.close();
              }
x8             catch(Exception e)
                  {
x9                System.out.println("Error in file");
                  }
      }
x10    pointcut withdrawlog(int amnt):call(void Account.withdraw(int))&&args(amnt);
x11   before( int amnt): withdrawlog1(amnt)
      {
       try{
x12          PrintWriter writer = new PrintWriter ("D:\\SDG\\log.txt");
x13          writer.println("withdraw amount="+amnt);
x14         writer.close();
          }
```

Figure 5.9: *Aspect part of AOP*

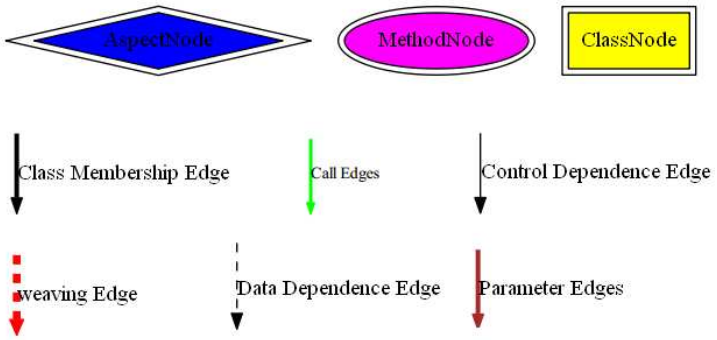

Figure 5.10: *Notations used*

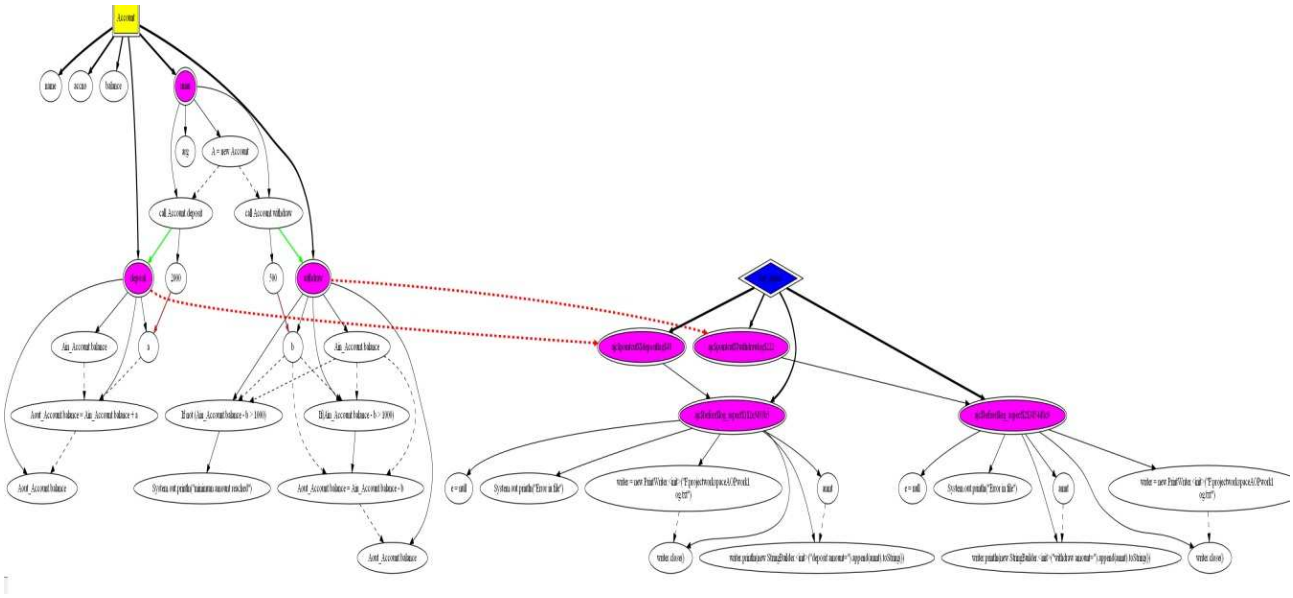

Figure 5.11: *SDG*

5.4 Compute Slicing of AOP

This section discuss the slicing of the previously generated JSDG. Here we have taken two pass algorithm[2] to compute static slice. Our tool takes one input slicing criterion i.e statement no. show that, it slices using the JSDG by considering that slicing criterion. Now we discuss bellow the Algorithm with taking previous example.

5.4.1 Algorithm

Two phase Approach

- The traversal in **pass one** starts from desired vertex and goes backwards along all edges except parameter-out edges.
- The traversal in **pass two** starts from all vertices reached in pass one and goes backwards along all edges except call and parameter-in edges.
- The slice is the union of 2 sets of vertices.

5.4.2 Case study for the Static Slicing of AOP

we have taken the same previous Aspect-oriented program shown in section 5.3.1, it has two part one Aspect part and other one non-Aspect part then we creates SDG for that two part separately and then join these two by using weaving edge. after creating SDG of that AOP we use two phase Approach to compute slice with criterion $< x13, amnt >$, which is shown the figure with shaded portion.

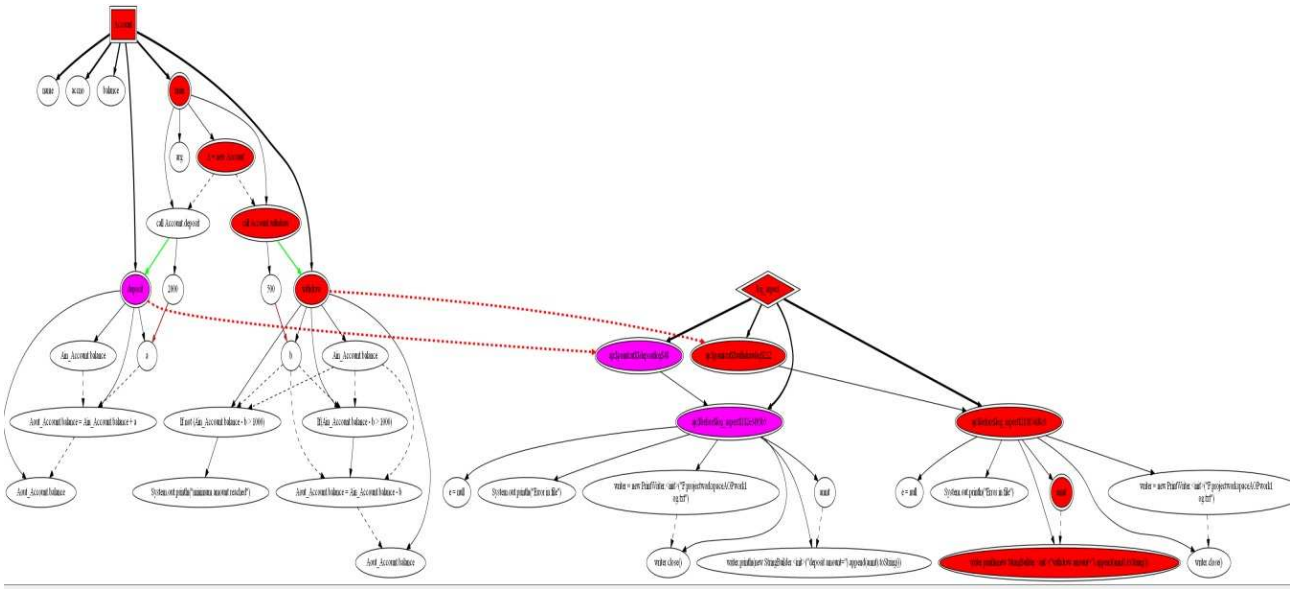

Figure 5.12: *Sliced with criterion* $< x13, amnt >$

5.5 Results

We have taken some open source Examples of OOP and AOP java program and compute slice of them then got the following results shown in the Table 5.1

Table 5.1: Performance

Programs	classes	Aspect	Time to generate SDG(ms)	Time to Slice(ms)
Binary Search Tree	2	nil	110	08
Stack implementation	2	nil	73	03
Doubly Linked List	2	nil	91	05
Account	1	1	111	12

Chapter 6

Conclusion and future work

- We have generated System Dependence Graph of Object-Oriented programs with all features,then compute static Slice using Two phase Approach[2].and Dynamic slice using Node Marking Approach.
- We have taken some Open source Example of java program for slicing using Our this Approach.
- Observed the time taken for constructing SDG and Slice.
- Our future work is to compute slice for Dynamic Slicing of AOP programs using an efficient Algorithm, as well as compute slice of Distributed-OOP and Concurrent-OOP.

Bibliography

[1] M. Weiser, " Program Slicing", Proceedings of the 5th International Conference on Software Engineering, San Diego, California, United States, March 09-12, 1981, pages 439-449.

[2] S. Horwitz, T. Reps and D. Binkley, "Inter-procedural Slicing using Dependence Graphs",

[3] Agrawal, Hiralal, and Joseph R. Horgan. "Dynamic program slicing." ACM SIGPLAN Notices. Vol. 25. No. 6. ACM, 1990. ACM Transactions on Programming Languages and Systems, 1990, pages 2661.

[4] D. W. Binkley and K. B. Gallagher, "Program slicing", Advances in Computers, Vol-43, 1996, pages 1-50.

[5] Walkinshaw, Neil, Marc Roper, and Murray Wood. "The Java system dependence graph." Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on. IEEE, 2003.

[6] Mohapatra Durga Prasad, Rajib Mall, and Rajeev Kumar. " Node-Marking Technique for Dynamic Slic- ing of Object-Oriented Programs", In Proceedings of Conference on Software Design and Architecture (SODA'04), 2004, pages 155-160.

[7] Mohapatra, D. P., Mall, R., and Kumar, R. (2006). An overview of slicing techniques for object-oriented programs. Informatica (Slovenia), 30(2), 253-277.

[8] Chen, Z., and Xu, B. Slicing object-oriented Java programs. ACM SIGPLAN Notices 36 (2001), 33-40.

[9] Tip, Frank. "A survey of program slicing techniques." Journal of programming languages 3.3 (1995): 121-189.

[10] MOHAPATRA D. P. Dynamic slicing of object oriented programs, PhD paper ,IIT Kharagpur 2005.

[11] Korel, Bogdan, and Janusz Laski. "Dynamic program slicing." Information Processing Letters 29.3 (1988): 155-163.

[12] Binkley, David W., and Keith Brian Gallagher. "Program slicing." Advances in Computers 43 (1996): 1-50.

[13] Zhao, Jianjun, and Martin Rinard. "System dependence graph construction for aspect-oriented programs." 7HFKQLFDO 5HSRUW 0 7 (2003).

[14] Zhao, Jianjun. "Slicing aspect-oriented software." Program Comprehension, 2002. Proceedings. 10th International Workshop on. IEEE, 2002.

[15] Korel, Bogdan, and Janusz Laski. "Dynamic program slicing." Information Processing Letters 29.3 (1988): 155-163.

[16] De Lucia, Andrea. "Program Slicing: Methods and Applications." scam. 2001.

[17] Harman, Mark, and Robert Hierons. "An overview of program slicing." Software Focus 2.3 (2001): 85-92.

[18] Binkley, David. "The application of program slicing to regression testing." Information and software technology 40.11 (1998): 583-594.

[19] Madhusmita, and Durga Prasad Mohapatra. "A node-marking technique for dynamic slicing of aspect-oriented programs." Information Technology,(ICIT 2007). 10th International Conference on. IEEE, 2007.

[20] http://www. graphviz. org/Documentation/dotguide. pdf, 2006.

[21] Ray, Abhishek, Siba Mishra, and Durga Prasad Mohapatra. "A Novel Approach for Computing Dynamic Slices of Aspect-Oriented Programs." arXiv preprint arXiv:1403.0100 (2014).

[22] Mohapatra, Durga Prasad, et al. "Dynamic slicing of aspect-oriented programs." Informatica 32.3 (2008): 261-274.

[23] Kiczales, Gregor, et al. Aspect-oriented programming. Springer Berlin Heidelberg, 1997.

[24] Sasirekha, N., and M. Hemalatha. "PROGRAM SLICING TECHNIQUES AND ITS APPLICATIONS." International Journal of Software Engineering and Applications 2.3 (2011).

[25] Liang, Donglin, and Mary Jean Harrold. "Slicing objects using system dependence graphs." Software Maintenance, 1998. Proceedings., International Conference on. IEEE, 1998.

[26] ter Braak, Timon. "Extending Program Slicing in AspectOriented Programming with InterType Declarations." 5th Twente Student Conference on IT. 2006.

[27] http://www4.comp.polyu.edu.hk/ cscllo/teaching/SDGAPI/